Application of RT-DEVS in Military

Mohammad Moallemi, Gabriel Wainer, Antoine Awad
Dept. of Systems and Computer Engineering, Carleton University, Centre of Visualization and Simulation (V-Sim)
1125 Colonel By Dr. Ottawa, ON, Canada.
{moallemi, gwainer}@sce.carleton.ca, aawad3@connect.carleton.ca

Dieynaba Alpha Tall,
Polytech Marseille – Departement de Genie Industriel
Domaine Universitaire de St Jérôme,
13397, MARSEILLE Cedex 20,
France
dieynaba-alpha.tall@etu.univ-provence.fr

Keywords: Discrete event simulation, DEVS, Military application development, Real-Time Simulation and Control

Abstract

Today, model based simulation is a popular scheme for simulating real world events. There has been some effort to use the same models that have been developed for simulation purposes for control applications. This approach permits model reuse and reliability for critical embedded control applications. In this paper a simulation model of an autonomous robot has been used to control a simulated reconnaissance vehicle on battle field. Since it is very costly to construct a real battle field situation and to verify the performance of military devices, model continuity from simulation to embedded control is a cost-effective and easy process for developing military applications. We have used DEVS (Discrete EVent System specification) formalism to define the robotic vehicle model and conducted variety of tests by simulation variety of scenarios. The final model has been embedded on a tank shaped robot.

1. INTRODUCTION AND MOTIVATION

Military application development is a very critical and expensive task in the engineering field. Verification and testing phase is also more critical as it is very difficult to practically simulate a battle field situation in reality. Testing military equipments in a real battle field condition would be very costly and risky. Instead, computer simulation methods can be a good replacement for real environment testing. Simulation proposes a cost-effective and easy way of modeling the real world events and calibrating different conditions in a virtual environment.

Formal methods are safe and easier techniques for modeling the environment under focus. Different formal methodologies have been presented in the area of modeling and simulation (M&S). Many state-based approaches, such as Verilog [1], VHDL [2], Petri Nets and Timed Petri Nets [3], Timed Automata [4], State Charts [5] and Finite State Machines [6] have been presented for M&S.

Model continuity from simulation to real-time embedded control application eases both design and verification phases of military control application development. The simulated models are tested in a risk free environment, and the transportation of these models to the real hardware environment is also straight forward. Deploying a formal methodology for the design phase and using an already available development tool speeds up application development and increases the reliability of the final product. Using a previously tested simulation model makes it portable on any hardware platform. Model reuse leads to a faster and reusable model development process as many already available sub-models can be integrated with the new ones. Formal techniques usually provide graphical representation for a model, helping model developer to better conceive the model behavior. Graphical representation also speeds up the design phase and assists in robust model development.

The cost and criticality of military application development is a major issue in choosing simulation methods to map battle field events onto virtual environment. The goal here is not only having a visual simulation of the battle field, but also developing a robust formal model of the control application that would be used on the real hardware platform. Several critical issues are brought up in real-time applications which can be verified with real-time simulation. The ability to simulate the model in virtual-time and real-time is a major motivation for this research. Portability of the simulated model on different hardware platforms lets us test the model on small scale robotic hardware.

2. BACKGROUND

Discrete EVent System specification (DEVS) [7] is a sound formal framework based on generic dynamic systems which provides a well-defined coupling of components and construction of hierarchical and modular systems. It also supports discrete event approximation of continuous systems and repository reuse. DEVS theory provides a rigorous methodology for representing models, and it does

present an abstract way of thinking about the world with independence of the simulation mechanisms, underlying hardware, and middleware. A real system modeled with DEVS is described as a composite of sub-models, each of them being behavioral (atomic) or structural (coupled).

A DEVS atomic model is formally defined by:

$M = <X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta>$, Where:

$X = \{(p,v) \mid p \in IPorts, v \in Xp\}$ is the set of input ports and values;

$Y = \{(p,v) \mid p \in OPorts, v \in Yp\}$ is the set of output ports and values;

S: is the set of sequential states;

δ_{int}: $S \rightarrow S$ is the internal state transition function;

δ_{ext}: $Q \times X \rightarrow S$ is the external state transition function, where:

$Q = \{(s,e) \mid s \in S, 0 < e < ta(s)\}$ is the total state set, e is the time elapsed since the last state transition;

λ: $S \rightarrow Y$ is the output function;

ta: $S \rightarrow R^{+}_{0,\infty}$ is the time advance function.

The semantics for this definition is given as follows. At any time, a DEVS coupled model is in a state $s \in S$. In the absence of external events, the model will stay in this state for the duration specified by ta(s). When the elapsed time $e = ta(s)$, the state duration expires and the atomic model will send the output $\lambda(s)$ and performs an internal transition to a new state specified by $\delta_{int}(s)$. Transitions that occur due to the expiration of ta(s) are called internal transitions. However, state transition can also happen due to arrival of an external event which will place the model into a new state specified by $\delta_{ext}(s,e,x)$; where *s* is the current state, *e* is the elapsed time, and *x* is the input value. The time advance function ta(s) can take any real value from 0 to ∞. A state with ta(s) value of zero is called transient state, and on the other hand, if ta(s) is equal to ∞ the state is said to be passive, in which the system will remain in this state until receiving an external event. A DEVS coupled model is composed of several atomic or coupled sub-models and their couplings.

Figure 1 shows a hierarchical DEVS model. This model is composed of two atomic models (Generator, Buffer and Processor) and two coupled models: the top model that contains generator atomic model and BUF-PROC coupled mode, the BUF-PROC coupled model includes two atomic models: BUF and PROC. The port connections are also visible in the figure. For example the output port "out" of atomic model PROC is connected to the "done" input port of BUF atomic model within the same coupled model and also is connected to the output port of the its parent coupled model which connects this output to the Top model output port.

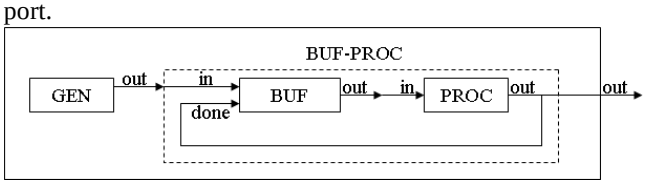

Figure 1. Generator-Buffer-Processor hierarchical DEVS model

Embedded CD++ (E-CD++) [8] is an extension to CD++ [9] toolkit that has been developed based on Parallel DEVS (P-DEVS) [10] formalism which has converted the virtual time function of CD++ into a real-time function (using a time advance function tied to the real-time clock) and added hardware interaction capability to it. Model implementation in E-CD++ is done by writing C++ code in a text-based Linux environment with open source tools. In order to improve the development and simulation experience, an IDE is provided for the E-CD++ simulator as an Eclipse plug-in that contains E-CD++ functionalities. It also has a graphical model designer that supports GGAD (Generic Graphical Advanced environment for DEVS modeling and simulation) diagram [11].

3. AUTONOMOUS RECONNAISSANCE VEHICLE

Detective robot model is a sophisticated DEVS-based system which is able to explore battle field for any suspicious object. Using DEVS as a formal method for designing such a model provides a formal paradigm for military application design and reduction of application development cost. This methodology enabled us to design this model and test it using a small handcrafted robot in the lab. Figure 2 shows different views of the reconnaissance vehicle.

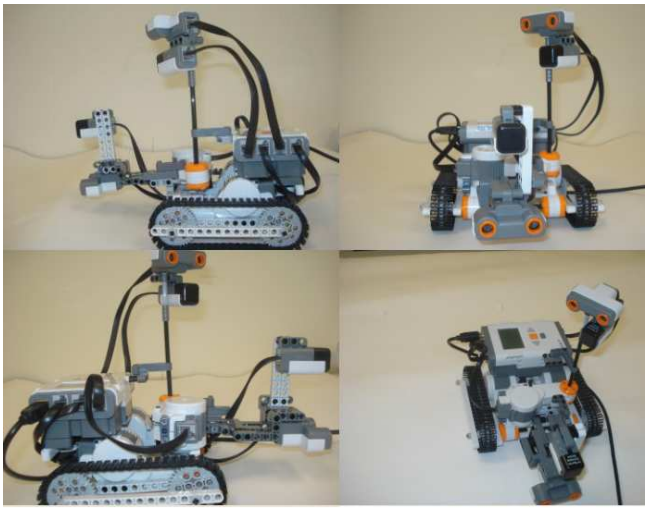

Figure 2. Autonomous reconnaissance vehicle

The robot vehicle uses a radar device mounted on top of it which scans the area surrounding it. The radar is equipped with a sonar (ultra sonic) sensor and a compass sensor. The function of the sonar sensor is to detect an object and the compass sensor is responsible to detect the direction of the obstacle to report it to the movement controller of the robot. In a real scale design, the sonar sensor can be replaced with a more sophisticated one. There is another sonar sensor mounted on the robot body accompanied by another compass sensor. The idea behind this is to receive the direction angle of the object detected by the radar and steer the robot towards the target. The sonar sensor on the robot body helps avoiding obstacles in front of it in order to find the target. There is a motor mounted on the robot to turn the radar to accomplish 180 degrees clockwise and counterclockwise turns for scanning the area around. There are two more motors connected to the robot used for moving and steering.

3.1. DEVS Model Specifications

Figure 3 illustrates DEVS model hierarchy defined for the robotic vehicle model.

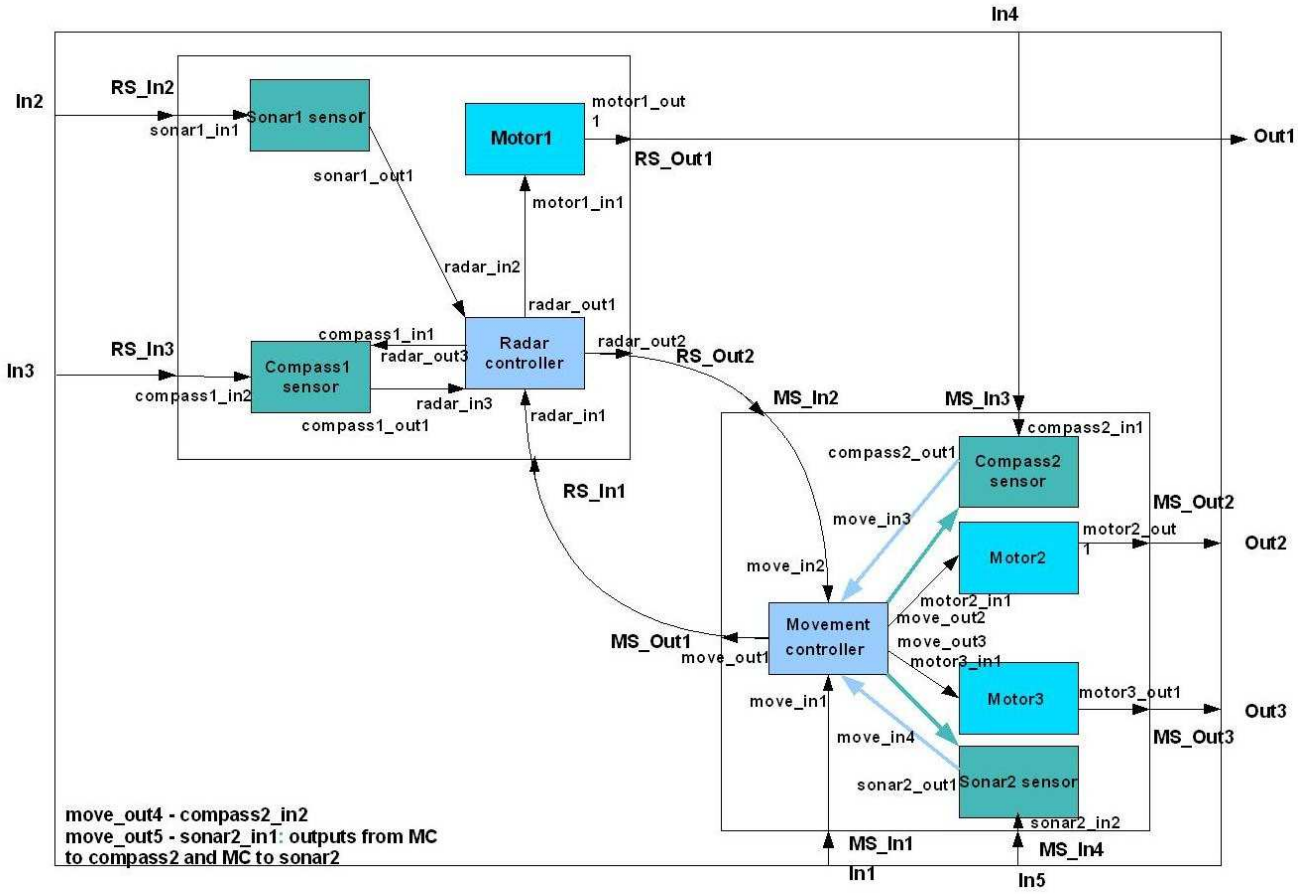

Figure 3. Detective robot DEVS model hierarchy

The model is constructed of a Top coupled model which contains two coupled models: *Radar System* and *Movement System*. The *Radar System* is responsible for controlling the radar hardware behavior. It is composed of four atomic models: *Sonar1 sensor*, *Compass1 sensor*, *Motor1* and *Radar controller*. *Radar controller* is the main atomic model in *Radar System* coupled model which synchronizes the other three atomic models in the coupled model and talks to *Movement System*. As soon as the simulation/execution starts, *Radar controller* model receives a signal from *Movement System* coupled model; the former sends a signal to *Motor1* model ordering the spinning action. *Motor1* model performs the simple job of spinning the radar motor clockwise or counterclockwise. *Radar controller* periodically changes the direction of spinning of *Motor1* model so that the radar turns 180 degrees clockwise and 180 degrees counterclockwise, scanning a certain radius surrounding the robot. *Sonar1 sensor* atomic model controls

the sonar sensor which is mounted on the radar. It starts receiving periodic inputs from the sonar sensor hardware after obtaining the start signal from *Radar controller* and forwards them to *Radar controller* model. As soon as the *Sonar1 sensor* detects an object in the scan radius area, it forwards a signal to the *Radar controller* model. The latter sends three outputs: 1) one to *Compass1 sensor,* ordering the detection of the direction (based on the angle of deviation from the north direction) of the object. 2) A signal to the *Motor1* to stop spinning. 3) A signal to the *Movement System* to stop the robot. Once *Compass1 sensor* model receives this signal, catches the angle from the sensor device and reports the value back to the *Radar controller*. The latter forwards this value to *Movement System.*

DEVS formal definition of *Radar controller* model is as follows:

$M = <X, S, Y, \delta_{ext}, \delta_{int}, \lambda, ta>$, Where

X: (radar_in1, 1) Connected to *Movement System* start signal. (radar_in2, 1) connected to *Sonar1 sensor* object detection signal. (radar_in3, degree) connected to *Compass1 sensor* angle of the object.

S: ”Idle”, “Prepare for Working”, “Ask motor1 to turn right”, “Ask motor1 to turn left”, “Ask motor1 and MC to stop”, “Wait compass1 degree”, “Send mess. degree to MC”.

Y: (radar_out1, 0, 1, 2) Connected to the *Motor1*, 0: stop, 1: spin clockwise, 2: spin counterclockwise. (radar_out2, 1, degree) connected to *Movement System*, 1: stop signal, degree: target degree, (radar_out3, 1) connected to *Compass1 sensor* angle detection signal.

δ_{ext}: Receives inputs from the input port and initiates appropriate state transitions.

δ_{int}: defines state changes based current state.

λ: based on the input value and the current state sends the output signals to the output ports.

ta: real-time advance function.

Figure 4 illustrates the GGAD state diagram of the *Radar controller* atomic model. Note that each circle indicates a state and the continuous line connections show external transitions and dashed lines show internal transitions between states. The labels on external transitions show the input port and input value used by δ_{ext} function and labels on internal transitions show output port and output value produced by λ function before δ_{int} function. The duration of each state (ta(s)) is indicated in the circle.

Movement System coupled model is composed of five atomic models:

1) *Motor2*: Controls the function of the right motor which is connected to the right robot tread. Carries out the same functions as *Motor1* model.

2) *Motor3*: Controls the function of the left motor, connected to the left robot tread. Carries out the same functions as *Motor1* and *Motor2* models.

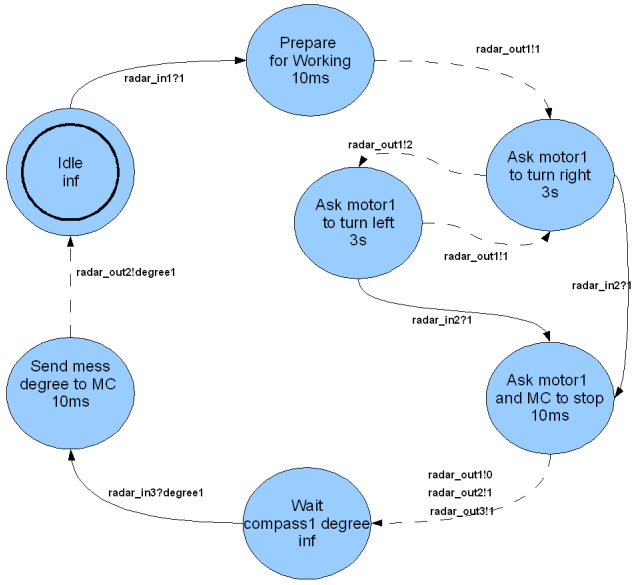

Figure 4. *Radar controller* GGAD diagram

3) *Compass2 sensor*: Controls the compass sensor mounted on the robot robot. It reports the current direction of the robot to the *Movement controller* model. It also matches the angle which the robot is heading with the angle that the target is located while the robot is turning towards the target.

4) *Sonar2 sensor*: controls the sonar sensor mounted on the robot. The main function of this atomic model is to find the target when the robot is heading towards it.

5) *Movement controller*: synchronizes the other five atomic models in *Movement System* coupled model. A predefined event coming from the event file to this model from input port *In4* fires the simulation/execution process. Once this input is received, *Movement controller* starts both motors to move forward by sending signals to *Motor2* and *Motor3* models and also the radar. As soon as it receives a signal from *Radar controller* confirming detection of an object, the former stops both motors and waits to obtain the target angle from the latter. After receiving the angle, *Movement controller* sends a signal to *Compass2 sensor* to catch the current robot heading angle. Using these two values it calculates which direction to turn, then orders both motors to spin in a direction which accomplishes the calculated turning direction. It also informs the *Compass2 sensor* to start capturing the robot heading direction in a periodic manner and comparing it with the target direction. Once these two directions are matched *Compass2 sensor* informs *Movement controller* and the latter stops motors and orders them to move forward towards the target. While the robot is traveling towards the target, *Movement*

controller listens to *Sonar2 sensor* to find out when it is close enough to the target to stop.

DEVS formal definition of *Movement controller* model is as follows:

M = <X, S, Y, δ_{ext}, δ_{int}, λ, ta>

X: (move_in1, 1) Connected to event file to receive start signal. (move_in2, 1, degree1) Connected to *Radar controller*, 1: object detection signal, degree1: target angle. (move_in3, 1, degree2) Connected to *Compass2 sensor*, 1: target angle and robot angle matched signal, degree2: robot heading angle. (move_in4, 1) Connected to *Sonar2 sensor*, target detection signal.

S: ”Stop”, “Prepare to move forward”, “Moving forward”, “Ask MC to stop”, “Wait radar mess”, “Prepare to receive”, “Wait to receive”, “Prepare to turn”, “Turn”, “Ask to prepare to Move forward”, “Move forward”, “Ask to prepare to stop”.

Y: (move_out1, 1) Connected to the *Radar controller*, start signal. (move_out2, 0, 1, 2) Connected to *Motor2*, 0: stop, 1: spin clockwise, 2: spin counterclockwise. (move_out3, 0, 1, 2) Connected to *Motor3*, 0: stop, 1: spin clockwise, 2: spin counterclockwise. (move_out4, 1, degree1) Connected to *Compass2 sensor*, 1: catch tang heading angle, degree1: match target angle with robot angle. (move_out5, 1) connected to *Sonar2 sensor*, target detection signal.

δ_{ext}: Receives inputs from the input port and initiates appropriate state transitions.

δ_{int}: defines state changes based current state.

λ: based on the input value and the current state sends the following outputs signals to the output ports.

ta: real-time advance function.

Figure 5 illustrates the GGAD state diagram of the *Movement controller* atomic model.

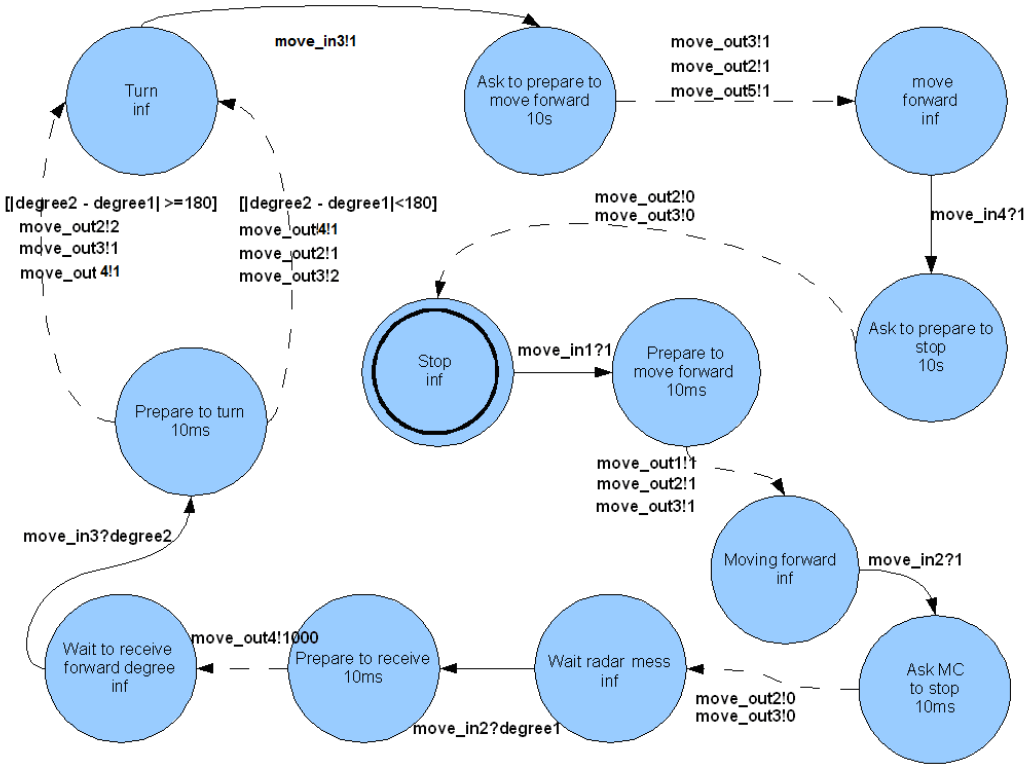

Figure 5. GGAD state diagram of Movement controller

3.2. Implementation on E-CD++

E-CD++ provides a simple framework for programming DEVS models. DEVS model hierarchical structure and couplings are provided in a model file with a specific format. DEVS main three functions δ_{ext}, δ_{int}, λ are overridden by user in C++ language. This framework speeds up implementation process, and improves reliability of the final product to a high extent. The following code illustrates the model file of the detective robot model.

```
1  [top]
2  components : RadarSystem MovingSystem
3  out : Out1 Out2 Out3
4  in : In1 In2 In3 In4 In5
5  Link : In1 MS_In1@MovingSystem
6  Link : In2 RS_In2@RadarSystem
```

```
7  …
8  [RadarSystem]
9  components        :         sonar1@Sonar1
   compass1@Compass1           motor1@Motor1
   radarcontroller@RadarController
10 in : RS_In1 RS_In2 RS_In3
11 out : RS_Out1 RS_Out2
12 Link : RS_In1 radar_in1@radarcontroller
13 Link : RS_In2 sonar1_in1@sonar1
14 …
15 [MovingSystem]
16 components : motor2@Motor2 motor3@Motor3
   movementController@MovementController
   compass2@Compass2 sonar2@Sonar2
17 in : MS_In1 MS_In2 MS_In3 MS_In4
18 out : MS_Out1 MS_Out2 MS_Out3
19 Link : MS_In1 move_in1@movementController
20 Link : MS_In2 move_in2@movementController
21 …
```

Line 1 starts the definition of Top model which contains two coupled components in line 2, *Radar System* and *Movement System*. Lines 3 and 4 declare output and input ports of the Top model. Lines 5 and 6 declare the internal couplings between the two coupled components inside the Top model. The same logic is repeated for the other two coupled models *Radar System* and *Movement System*.

The following code snippet shows a portion of the three DEVS functions implementation for *Radar controller* atomic model.

```
1  Model &RadarController::externalFunction( const ExternalMessage &msg )
2  {
3    if(state==Idle && msg.port() == radar_in1){
4      if(msg.value()==1){
5        state = Pr_Wrk;
6        holdIn( Atomic::active, radarTime );
7      }
8    }
9    …
10   return *this;
11 }

12 Model &RadarController::internalFunction( const InternalMessage & )
13 {
14   switch (state){
15     case Pr_Wrk:
16       state = Trn_Right;
17       holdIn( Atomic::active, turnTime );
18       break;
19   …
20   }
21   return *this;
22 }

23 Model &RadarController::outputFunction( const InternalMessage &msg )
24 {
25   switch (state){
26     case Pr_Wrk:
27       sendOutput( msg.time(), radar_out1, 1) ;//working
28       break;
29   …
30   };
31   return *this ;
32 }
```

Lines 1 to 11 show a portion of the code for δ_{ext} function in which the input value 1 from port “radar_in1” is being checked while the model is in “Idle” state and the state changes to “Prepare for Working”. The holdIn function in E-CD++ sets the time duration (ta(s)) of the state which is declared in model file. Line 12 to 22 show part of the δ_{int} function, which maps the dashed lines of the GGAD diagram. In this code the transition from state “Prepare for Working” to “Ask motor1 to turn right” is implemented. Lines 23 to 32 implement part of the λ function, which is producing output of the “Prepare for Working” state.

Code similar to this paradigm is used for the other atomic models in the system.

3.3. Simulation Results

As mentioned before, E-CD++ provides virtual-time and real-time simulation framework for DEVS models as well as tools to add a hardware driver to the model to embed and execute it on the hardware. We ran variety of simulation scenarios both in virtual-time and real-time using the detective robot model. Figure 6 shows the event file of a simulation scenario versus the result of the simulation in E-CD++.

Event file

```
00:00:02:00   In1   1
00:00:05:00   In2   1
00:00:06:00   In3   200
00:00:09:00   In4   50
00:00:09:50   In4   90
00:00:10:00   In4   150
00:00:10:50   In4   180
00:00:11:00   In4   190
00:00:11:50   In4   200
00:00:15:00   In5   2
```

Simulation

```
00:00:02:020 out2 1
00:00:02:020 out3 1
00:00:02:030 out1 1
00:00:05:030 out1 0
00:00:05:040 out2 0
00:00:05:040 out3 0
00:00:09:030 out2 1
00:00:09:030 out3 2
00:00:11:080 out3 1
00:00:15:030 out2 0
00:00:15:030 out3 0
```

Figure 6. Snapshot of E-CD++ event file and output file for detective robot model

The first line of the *Event file* shows the value 1 which is passed to the input port "In1" of the Top model after 2 seconds of the start of the simulation. Event file only accepts inputs to the Top coupled model ports. Port "In1" is hierarchically connected to the *Movement controller* atomic model; therefore this input is forwarded to this atomic mode and triggers the start of the simulation. Respectively, the first three lines in the outputfile are the outputs to the three motors (two robot motors and one radar motor) which are produces 20 to 30 milliseconds later based on the state durations. After the robot starts to move and the radar to scan, at the fifth second (line 2 of event file) an input is forwarded to port "In2" which is connected to *Sonar1 sensor* model. Hence, the all three motors are stopped (line 3 to 6 of output file) and the *Radar controller* waits for the target angle from *Compass1 sensor*. The angle of 200 degrees is injected to the model at the 6th second and this triggers *Movement controller* to obtain the robot current heading angle. 50 degrees angle value is entered to port "In4" which is connected to *Compass2 sensor* atomic model and the latter forwards it to *Movement controller*, where the calculation of the turning direction happens and the outputs are shown in lines 7 and 8 of the output file. Once, one motor of the robot spins clockwise and the other one counterclockwise, the robot starts turning to either of the directions. After that, 5 inputs are entered to port "IN4" which simulate the periodic angle inputs while the robot is turning towards the target. As soon as 200 degrees is detected by *Compass2 sensor* model, the model notifies *Movement controller* model about this match and the latter orders both motors to move forward (only one motor which is not spinning clockwise will be ordered to spin clockwise). Finally the target detection signal is injected to port "In5" which is connected to *Sonar2 sensor* which notifies *Movement controller* and the latter stops both motors.

After building the robotic vehicle, the hardware driver was written for the connected ports and the model was run on the robot. Several tests have been carried out from different angles and the robot could pass all of them successfully. Two videos of the robot are provided online in [12], [13].

4. CONCLUSIONS

DEVS as a formal modeling and simulation methodology provides risk-free and easy-to-modify battle field simulation strategies where the validity of the system is guaranteed. In this work, a DEVS-based real-time and embedded control model was introduced for an autonomous reconnaissance robotic vehicle in the battle field. We have presented the DEVS model specifications for the robotic vehicle and discussed the implementation details of the model in E-CD++. The Simulation results were elaborated to explain the functionality of the vehicle. Also, to illustrate the functional behavior of the robotic vehicle, sample simulation scenarios were captured on videos and links were provided to clearly observe how the system works. The use of a formal method like DEVS improved reliability and portability of the model on different hardware platforms which enabled us to verify the model using virtual-time and real-time simulations. Model reuse feature of DEVS eases the use of many parts of this model for developing a more sophisticated and real control model of a military autonomous vehicle. This work also presented how the E-CD++ toolkit provides an object oriented framework for programming DEVS-based speeding up the implementation phase and improving robustness of the final product.

Reference

[1] Thomas, Donald, Moorby, Phillip "The Verilog Hardware Description Language" Kluwer Academic Publishers, Norwell, MA. ISBN 0-7923-8166-1.

[2] Peter J. Ashenden "The VHDL Cook Book" July, 1990, available on line at: http://www.comms.scitech.susx.ac.uk/fft/vhdl/VHDL-Cookbook.pdf Accessed on Sep/16/2009.

[3] Petri Nets: Properties, Analysis and Applications, by Tadao Murata, in: Proceedings of the IEEE, vol. 77, no. 4, April 1989.

[4] R. Alur D.L.; Dill "A theory of timed automata", Theoretical computer science, Vol. 126, No 2, pp 183-235, 1994.

[5] D. Harel et al., "On the Formal Semantics of StateCharts", Proceedings of the Symposium on Logic in Computer Science, pp. 54-64, 1987.

[6] Wagner, F., "Modeling Software with Finite State Machines: A Practical Approach", Auerbach Publications, 2006, ISBN 0-8493-8086-3.

[7] B. Zeigler, T. Kim, H. Praehofer. "Theory of Modeling and Simulation". Academic Press 2000, ISBN-10: 0127784551.

[8] YU, J.; WAINER, G. "E-CD++: a tool for modeling embedded applications". In Proceedings of the 2007 SCS Summer Computer Simulation Conference. San Diego, CA. 2007.

[9] Wainer, G. "CD++: a toolkit to define discrete-event models". Software, Practice and Experience. Wiley. Vol. 32, No.3. pp. 1261-1306. November 2002.

[10] Chow A, Kim D, Zeigler B. "Parallel DEVS: A parallel, hierarchical, modular modeling formalism" In Proceedings of Winter Simulation Conference, 1994, Orlando, Florida.

[11] G. Christen, A. Dobniewski and G. Wainer, "Modeling State-Based DEVS Models in CD++". In Proceedings of MGA, Advanced Simulation Technologies Conference 2004 (ASTC'04). Arlington, VA. U.S.A.

[12] Detective Robot robot video 1 available online at: http://www.youtube.com/watch?v=w-bwwl4CP4c

[13] Detective Robot robot video 2 available online at: http://www.youtube.com/watch?v=61vXI9qujZI